OPEN ACCESS Freely available online

PLOS ONE

Compression of Structured High-Throughput Sequencing Data

Fabien Campagne[1,2]*, Kevin C. Dorff[1], Nyasha Chambwe[1,2], James T. Robinson[3], Jill P. Mesirov[3]

1 The HRH Prince Alwaleed Bin Talal Bin Abdulaziz Alsaud Institute for Computational Biomedicine, The Weill Cornell Medical College, New York, New York, United States of America, **2** Department of Physiology and Biophysics, The Weill Cornell Medical College, New York, New York, United States of America, **3** The Eli and Edythe L. Broad Institute, Massachusetts Institute of Technology and Harvard University, Cambridge, Massachusetts, United States of America

Abstract

Large biological datasets are being produced at a rapid pace and create substantial storage challenges, particularly in the domain of high-throughput sequencing (HTS). Most approaches currently used to store HTS data are either unable to quickly adapt to the requirements of new sequencing or analysis methods (because they do not support schema evolution), or fail to provide state of the art compression of the datasets. We have devised new approaches to store HTS data that support seamless data schema evolution and compress datasets substantially better than existing approaches. Building on these new approaches, we discuss and demonstrate how a multi-tier data organization can dramatically reduce the storage, computational and network burden of collecting, analyzing, and archiving large sequencing datasets. For instance, we show that spliced RNA-Seq alignments can be stored in less than 4% the size of a BAM file with perfect data fidelity. Compared to the previous compression state of the art, these methods reduce dataset size more than 40% when storing exome, gene expression or DNA methylation datasets. The approaches have been integrated in a comprehensive suite of software tools (http://goby.campagnelab.org) that support common analyses for a range of high-throughput sequencing assays.

Citation: Campagne F, Dorff KC, Chambwe N, Robinson JT, Mesirov JP (2013) Compression of Structured High-Throughput Sequencing Data. PLoS ONE 8(11): e79871. doi:10.1371/journal.pone.0079871

Editor: Frederique Lisacek, Swiss Institute of Bioinformatics, Switzerland

Received April 10, 2013; **Accepted** September 27, 2013; **Published** November 18, 2013

Funding: This investigation was supported by grant UL1 RR024996 (National Institutes of Health (NIH)/National Center for Research Resources) of the Clinical and Translation Science Center at Weill Cornell Medical College, by grant LLS 6304-11 from the Lymphoma and Leukemia Society Translational Research Program, and by R01 MH086883 (NIH/National Institute of Mental Health). The Tri-Institutional Training Program in Computational Biology and Medicine provided support for NC. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors have declared that no competing interests exist.

* E-mail: fac2003@campagnelab.org

Introduction

Many scientific disciplines, including high-energy physics, astronomy and more recently biology, generate increasing volumes of data from automated measurement instruments. In biology, modern high-throughput sequencers (HTS) are producing a large fraction of new biological data and are being successfully applied to study genomes, transcriptomes, epigenomes or other data modalities with a variety of new assays that take advantage of the throughput of sequencing methods [1,2,3]. In addition to sequenced reads, data analyses yield secondary data, such as alignments of reads to reference genome. Sequencing throughput has more than doubled every year for the last ten years [4] resulting in storage requirements on the order of tens of terabytes of primary and secondary high throughput sequencing data in a typical laboratory. Major sequencing centers in the USA and worldwide typically require several tens of petabytes of storage to store reads and secondary data during the lifetime of their projects. Before study publication, read and alignment data are deposited in sequence archives to enable other groups to reanalyze the data. While improvements in sequencing throughput and experimental protocols continue to generate ever-larger volumes of HTS data, pressing questions remain. Namely, how to store these data to minimize storage costs, maximize computational efficiency for data analysis, increase network transfer speeds to facilitate collaborative studies, or to facilitate reanalysis or perusal of data stored in archives.

A popular method for storing read data is in FASTQ files [5]. Such files are text files typically compressed with GZIP or BZip2 compression and hold both nucleotide bases of the reads as well as quality scores. The latter indicate the reliability of each base call and are central to many analyses, such as genotyping. Compressed text files have critical problems: they are slow to parse, and they do not support random access to subsets of the reads, a feature that is critical to support parallelization of the alignment of the reads to a reference genome.

Most analyses require aligning read data to a reference genome, a process that yields HTS alignment data. When computed, HTS alignment data has traditionally been stored in a variety of file formats. Early text formats were quickly abandoned in favor of binary formats, and among these, the BAM (Binary Alignment/Map) format has become very popular and is now widely used [6]. A key problem with BAM is that the BAM format cannot be seamlessly adapted to support new applications. For instance, developers of TopHat fusion were not able to extend the BAM format to store information about gene fusions, and instead had to create a variant of the SAM text format (http://tophat.cbcb.umd.edu/fusion_manual.html#output). Developers of new analysis software based on BAM cannot seamlessly extend BAM for new applications because various programs that read/write BAM, developed worldwide, would need to be manually modified for

each change to the specification, a process that is all but practical. Another key weakness of the BAM format is that BAM files require approximately the same amount of storage as the unaligned reads, for each alignment represented in the format.

The CRAM format, developed for the European Nucleotide Archive, was developed to try and compress HTS alignments better than can be achieved with BAM. CRAM achieves strong compression of alignment files using custom developed compression approaches parameterized on characteristics of simulated HTS alignment data [7]. A key innovation of CRAM was the recognition that different applications need to preserve different subsets of the data contained in a BAM file. Preserving different subsets of the information can yield substantial storage savings for these applications that do not require all the data. However, CRAM shares a key weakness with BAM. Namely, it is unable to seamlessly support changes to the data format. Changes to extend the file format require manual coding and careful design of custom compression approaches for the new data to be stored. An additional problem is that CRAM cannot compress HTS alignments when they are not already sorted by genomic position. This is a significant problem because alignments are first determined in read order before they can they be sorted by genomic position. This problem limits the usefulness of the CRAM format to HTS archives, and prevents its use as a full replacement of the BAM format. We believe that these weaknesses are serious drawbacks because the HTS field is progressing very rapidly, sequencing throughput increases exponentially and new experimental advances often require extensions to the data schemas used to store and analyze the new types of data.

In summary, current approaches are unable to strongly compress HTS data while supporting the full life-cycle of the data, from storage of sequenced reads to parallel processing of the reads and alignments during data analysis to archiving of study results. In this manuscript, we present a comprehensive approach that addresses these challenges simultaneously and a robust software implementation of these methods.

Results

Overview

We have devised a novel approach for compressing HTS data. We benchmarked this approach against BAM and CRAM compression of HTS alignment data. The approach was integrated in a comprehensive software system, which includes the Goby framework (this manuscript http://goby.campagnelab.org), IGV [8], BWA [9] and GSNAP [10], and demonstrates proof of principle for compression, visualization and analysis of HTS data with these new approaches. The following sections describe these results.

Structured data schemas

We developed structured data schemas to represent HTS read and alignment data (Figure S1A in File S1) with Protocol Buffers technology (PB) [11]. PB automates reading and writing structured data and provides flexibility with respect to changes in the schemas. Extending the schema requires editing a text file and recompiling the software. The new software is automatically compatible with versions of the software that are unaware of the schema extension. PB schema flexibility therefore provides the means to evolve the file formats over time as a collaborative effort without breaking existing software.

Large datasets

We extended PB with methods to store large datasets and to define configurable compression/decompression methods (called codecs, see Methods). We developed codecs for general compression methods (PB data compressed with the GZip or BZip2 methods, Figure 1A), and a Hybrid codec that provides very strong compression of alignment data, while retaining the flexibility of PB schema evolution (see Figure S1B in File S1 and Methods). Finally, we group data in tiers according to the most likely use of each kind of data (Figure 2) and have developed a framework and a set of tools (see [12] and Methods) to support efficient computation with HTS data expressed in these formats. In the next sections, we describe our contributions to the compression of HTS alignment data.

General compression

General compression approaches are widely used and were developed to compress unstructured data (e.g., streams of bytes such as text in a natural language). The most successful general compression approaches employ probabilistic compression, where smaller sequences of bits are used to represent symbols with high probability in the input data, and longer sequences are used to represent symbols of lower probability (the mapping from sequences of symbols to sequences of bits is called a code). Arithmetic coding can yield near-optimal codes (i.e., streams of bits of length close to the theoretical lower bound), given a model of symbol probabilities. However, the question of how to construct effective probabilistic models of unstructured data is a difficult one because the models have to be inferred from observing the stream of unstructured data. Since the cost of inferring the model grows with model complexity, progress in compression ratios has often been obtained at the expense of compression speed.

Structured data compression

In this manuscript, we demonstrate that when presented with structured HTS alignment data (data organized according to a well-specified schema, see Figure S1A in File S1 for an example), it becomes possible to leverage the data structure to facilitate model inference. We have devised and present several such techniques: separate field encoding, field modeling, template compression, and domain modeling (see illustration of these new methods on Figure 1B-E).

Separate Field Encoding

This encoding reorganizes the dataset into lists of field values (Figure 1B). Where a traditional approach to compressing structured data often applies a general compression method to a serialized stream of structured data (Figure 1A), we reasoned that compression could benefit from inferring a model for each field of a data structure separately (Figure 1B). Leveraging the structure of the data makes it possible for model inference to detect regularities in successive values of the same field. Field encoding requires compressing blocks of messages together. We call each such block a chunk of PB data and typically encode 10,000–100,000 messages per block.

Field modeling

This technique is useful when the value of one field can be calculated or approximated from the value of another field of the same data structure (Figure 1C). In this case, this approach stores the difference between the approximated value and the actual value.

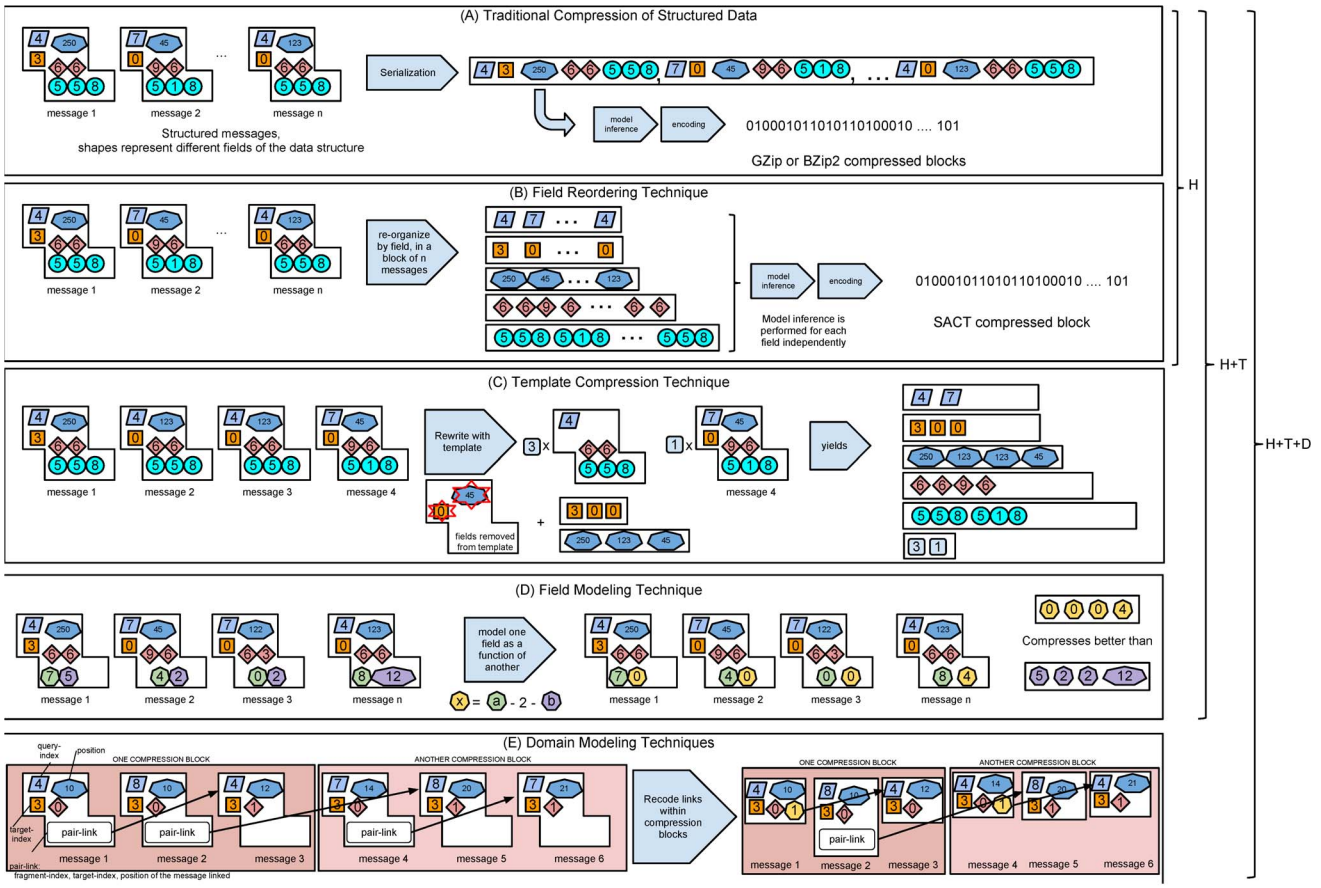

Figure 1. Structured Data Compression Techniques. We present the techniques that we devised for compressing structured High-Throughput Sequencing (HTS) data. We use a combination of general compression techniques (panel A) and of techniques that take advantage of the information provided by a data schema (B-E). (A) General compression techniques convert structured data to streams of bytes (serialization, typically done one message at a time) and then compressing the resulting stream of bytes with a general purpose compression approach such as Gzip and Bzip2. We use such techniques alone (Gzip and Bzip2 codecs) or in combination with structured data compression (Hybrid codecs, labeled H, H+T or H+T+D according to the technique used). (B) Separate field encoding reorganizes blocks of messages in lists of field values before compressing each field independently. The technique requires compressing blocks of PB messages, or PB chunks. (C) Field Modeling helps compress data by expressing the value of one field as a function of other fields and constants. (D) Template Compression Technique. Here, the data structure is used to detect subset of messages that repeat in the input messages. Fields that vary frequently are ignored from the template. The template values are stored with the number of template repetitions and the values needed to reconstruct the input messages. (E) Domain Modeling Technique. Alignment messages refer to each other with message links (i.e., references between messages) represented here as pair-link messages with three fields: position, target-index and fragment-index of the linked message. We realized that within a PB chunk, it is possible to remove the three fields representing the link and replace them with an integer index that counts how many messages up or down stream is the linked message in the chunk. Links from an entry in a chunk to an entry in another chunk cannot be removed and are stored explicitly with the three original fields.
doi:10.1371/journal.pone.0079871.g001

Template compression

This technique detects that a subset of a data structure (the template) repeats in the input (Figure 1D). It then stores the template, the number of times the template repeats and separately the fields that differ for each repetition. Template compression can be thought of as a generalization of run-length encoding to structured data.

Domain modeling

This technique requires a human expert to conduct a detailed analysis of the input data. In the case of HTS data, we developed an efficient representation of references between messages stored in the same PB chunk (Figure 1E).

Evaluation

We sought to evaluate the effectiveness of these approaches to compress HTS data by comparing them with general compression approaches (Gzip and BZip2) and with the BAM and CRAM approaches. To this end, because the compression effectiveness of any compression approach is expected to vary with input data, we assembled a benchmark of ten different datasets spanning several popular types of HTS assays: RNA-Seq (single-end and paired-end reads), Exome sequencing, whole genome sequencing (WGS), two DNA methylation assays: RRBS (reduced representation bisulfite sequencing) and whole genome Methyl-Seq (see Benchmark datasets in Materials and methods and Table S1 in File S1). Briefly, we found that combining all these approaches into the method labeled H+T+D (see Figure 1, H: Hybrid approach, T: template compression, D: domain modeling) yields the most competitive compression for HTS alignment data.

General Compression Benchmark

We evaluated the H and H+T approaches against general compression approaches when storing HTS alignment data. Table 1

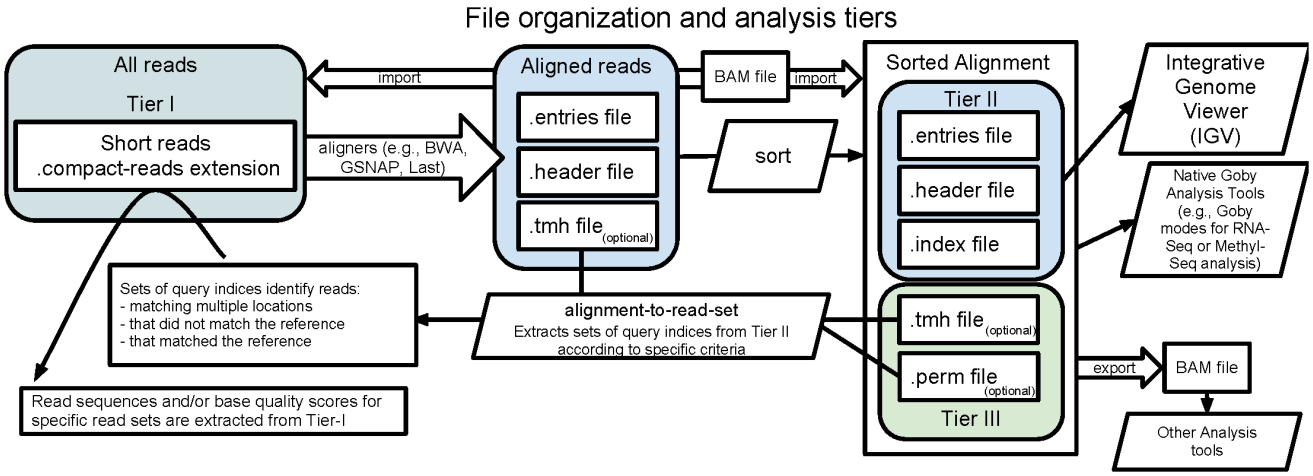

Figure 2. A comprehensive approach to store HTS data during the analysis life-cycle. This diagram illustrates how HTS data stored with the approaches described in this manuscript support common analysis steps of a typical HTS study. HTS reads (Tier I) are stored in files ending with the.compact-reads extension. These files can be read directly by alignment programs and facilitate efficient parallelization on compute grids. When the reads are aligned to a reference genome, alignment files are written in sets of two or three files. Files ending with.entries contain alignment entries. Each alignment entry describes how a segment of a read aligns against the reference genome. Files ending in.header contain global information about the reads, the reference genome, and the alignment (See Figure S1A in File S1 for the data schema that precisely describes these data structures). An optional.tmh file stores the identity of the reads that matched the reference so many times the aligner did not output matches for them. Aligned reads can be sorted with the 'sort' Goby tool, producing a.index file with enough information to support fast random access by genomic position. A permutation file (extension.perm) can also be produced to improve compression of sorted files (see Methods). Files in Tier II are stand-alone and can be transferred across the network for visualization (e.g., IGV). Files in Tier III are available for some specific types of analyses that require linking HTS alignments back to primary read data.
doi:10.1371/journal.pone.0079871.g002

presents the results of this benchmark. We tested the H and H+T variants of our approach (i.e., Figure 1B-D) and compared the evaluation metrics with those obtained when compressing the same data with the strong BZip2 general compression method [13,14]. We measured file sizes and compression effectiveness, as well as compression and decompression times (Table 1: comparison with BZip2, Table S2 in File S1: comparison to GZip) of the approaches compared. Each dataset is publicly available and all software required to replicate these results are freely available (see http://data.campagnelab.org/home/compression-of-structured-high-throughput-sequencing-data). Our benchmark results clearly establish that the H+T approach results in the smallest file sizes of either GZip or BZip2 compression. Compression ratios improve substantially when compared to BZip2 with faster compression speed. Decompression speed is only 36% slower than BZip2. These results are significant because BZip2 is often considered as one of the most effective

Table 1. Benchmark against BZip2 general compression.

	Storage Efficiency			Compression Times			Decompression Times		
Sample ID	**H**	**H+T**	**H+T+D**	**H**	**H+T**	**H+T+D**	**H**	**H+T**	**H+T+D**
HZFWPTI	60%	55%	42%	74%	80%	71%	138%	122%	138%
UANMNXR	59%	54%	41%	85%	88%	83%	139%	117%	135%
MYHZZJH	28%	24%	22%	40%	29%	34%	141%	101%	148%
ZHUUJKS	31%	26%	24%	39%	36%	34%	157%	110%	145%
EJOYQAZ	65%	56%	51%	51%	52%	52%	144%	119%	152%
JRODTYG	162%	88%	74%	46%	37%	42%	225%	95%	108%
ZVLRRJH	173%	96%	69%	93%	78%	93%	235%	119%	128%
XAAOBVT	95%	57%	46%	67%	70%	77%	120%	103%	138%
UCCWRUX	79%	51%	40%	79%	80%	70%	139%	114%	132%
HENGLIT	61%	59%	45%	94%	105%	104%	129%	113%	137%
Average	**81%**	**57%**	**45%**	**67%**	**66%**	**66%**	**157%**	**111%**	**136%**

Storage efficiency is calculated as the ratio of the size of compressed data with each method (H, H+T or H+T+D) vs BZip2 compressed data size, expressed as a percentage. A storage efficiency of 50% indicates that the specific method compressed the dataset to half the size of method BZip2 compression. Compression/Decompression time ratios measure the ratio of the time it takes a specific method to compress/decompress a dataset compared to the time it takes the BZip2 compression method for the same dataset. A ratio of 200% indicates that the specific method is twice slower than BZip2. See Fig. 1 for a description of the H, H+T and H+T+D methods.
doi:10.1371/journal.pone.0079871.t001

compression algorithms that remains sufficiently fast for practical use. Approaches that can achieve more than 10–15% better compression than BZip2 (i.e., 90–85% of the bzip2 baseline size, substantially larger than the results we report) are orders of magnitude slower for compression or decompression and are for this reason not practically useful for most applications. Here, we show that using the H+T+D approach (Hybrid codec with Template compression and domain specific optimizations, a lossless approach, see Figure 1E and Methods), we can compress structured alignment data to an average 45% of the size of BZip2 compressed data in 66% of the time needed by BZip2.

CRAM benchmark

Table 2 compares performance metrics for storing alignments with the Hybrid codec or with CRAM. We find that the method H+T compression yields files smaller than those obtained with the CRAM approach (average 73% of baseline). Interestingly, we noted that H+T compression is storing redundant information in the form of explicit forward and backward links between alignment entries (see pair links and splice links attributes in Figure S1A in File S1). The domain optimization technique described in Figure 1E, takes advantage of this observation and further increases compression of HTS alignment data (H+T+D approach) to an average 58% of the CRAM file sizes. Importantly, H+T+D appears to compress RNA-Seq and methylation HTS data much more effectively than CRAM (compressing these five datasets to 36% the size obtained with CRAM 2.0). Our results suggest that the CRAM approach may have been over-optimized to store WGS datasets to the extent that the optimizations become detrimental when compressing other types of HTS alignment data. Since the H+T+D approach adapts to the structure of the data, it does not suffer from this problem and yields substantial improvements when storing gene expression and epigenetic HTS datasets.

Compression Fidelity

When developing a new compression approach, it is critical to verify compression/decompression fidelity. We verified compression/decompression fidelity for the H+T and H+T+D approaches across all the benchmark datasets by comparing decompressed data to input data (see Methods). When tested in the same conditions, we found that CRAM 2.0 either failed to compress, or decompress, four out of the ten benchmark datasets (the CRAM 2.0 software crashed with an exception on these datasets). We previously had run the same benchmark comparing an earlier version of Goby to CRAM 0.7 (see pre-print http://arxiv.org/abs/1211.6664). CRAM 0.7 was able to compress and decompress all datasets, but with substantial data fidelity problems (http://arxiv.org/abs/1211.6664). Taken together, these observations suggest that CRAM 2.0 is still an experimental implementation of the approach published by Fritz et al [7] and indicates that the H+T+D approach is the current state of the art for compression of HTS alignment data.

HTS data analysis solutions

A compression approach such as H+T+D is only part of a solution to support efficient analysis of HTS data. A comprehensive approach must also provide support for storing reads, storing alignments during their analysis (including unsorted alignments), and most importantly, it must provide tools that support the data formats and make it possible to conduct data analysis with compressed data.

Comprehensive solutions for HTS data storage and analysis

Figure 2 presents the elements of a comprehensive approach to HTS data management. These elements include the Goby framework [12], and extensions to widely used HTS analysis programs such as the GSNAP [10] and BWA aligners [9], or the

Table 2. Benchmark against a CRAM baseline.

		Compression ratio A/B = size(A)/size(B)*100%		
		H+T/	H+T+D/	Notes
Sample ID	Kind	CRAM2.0 S2	CRAM2.0 S2	
HZFWPTI	Exome	86.48%	65.77%	
UANMNXR	Exome	87.85%	66.61%	
MYHZZJH	RNA-Seq	19.43%(!)	18.09% (!)	(!) CRAM 2.0 failed when decompressing this dataset.
ZHUUJKS	RNA-Seq	22.97%	20.95%	
EJOYQAZ	RNA-Seq	73.92%	67.77%	
JRODTYG	RRBS	34.04%	28.51%	
ZVLRRJH	Methyl-Seq	(*)	(*)	(*) CRAM 2.0 failed when compressing this dataset.
XAAOBVT	WGS	106.39%	86.43%	
UCCWRUX	WGS	117.84%(!)	92.49% (!)	
HENGLIT	WGS	103.98%(!)	79.78% (!)	
	Exome	87.17%	66.19%	
	RNA-Seq	38.77%	35.60%	
	bisulfite	34.04%	28.51%	
	WGS	109.40%	86.23%	
Average	All	72.54%	58.49%	

doi:10.1371/journal.pone.0079871.t002

Integrative Genomics Viewer [8]. Goby provides state of the art compression approaches, a rich application-programming interface to read and write HTS datasets and a program toolbox (see [12]). This toolbox can support the life-cycle of a HTS data analysis project, from alignment, to sorting and indexing the alignments by genomic position, to viewing aligned data in its genomic context with the Integrative Genomics Viewer.

Multi-tier data organization

These tools organize data in a multi-tier file organization. All reads (mapped or unmapped) are stored in Tier-I files and only differences between the reads and the reference sequence in alignment files (Tier II and III). This data organization reduces size considerably for Tier II alignments (in our benchmark up to 3–4% of the original BAM size for spliced RNA-Seq datasets, see Table 3), while keeping all the information that is typically stored in the BAM format in between the reads and alignment files. Combined, Tiers I and II capture all information currently stored in BAM files, but requires only 49% of the storage capacity needed by the now popular FASTQ/BAM storage scheme (Table 3). The benefit of a multi-tier organization increases with the number of alignments performed against the same set of reads (Table 3). See Methods for details about multi-tier data organization and further discussion of its advantages.

Discussion

We have presented new methods for storing, compressing and organizing HTS data. A key advantage of these methods is their ability to support seamless schema extensions, which makes it possible for data formats to evolve to meet the needs of the scientific community. These methods can be combined with compression methods to reduce the footprint of the datasets.

General compression methods such as GZip and BZip2 provide universal compression for any data schema. However, we have shown here that taking advantage of the structure of the data can yield state of the art compression of HTS alignment data. We anticipate that several of the techniques that we have introduced here can be generalized to arbitrary data schemas. For instance, separate field encoding, field modeling and template compression are techniques that could be used to develop fully automatic codec compilers (such compiler would analyze a data schema and yield a state of the art compression codec for the specific schema). We do not expect domain modeling to generalize to arbitrary schemas because it requires a detailed expertise about the data, but note that when used in combination with the other compression approaches described here, it has produced the new state of the art for compression of HTS alignment data.

Popitsch et al have recently included the Goby framework in a HTS compression benchmark [15]. Unfortunately, the setting used in that benchmark parameterized Goby to compress data with the Gzip compression codec (see [15] supplementary material where command lines used for Goby do not change compression from the default Gzip mode). The benchmark conducted by Popitsch et al therefore did not benchmark NGC against the state of the art Goby H+T+D approach available at the time the study was conducted. Data comparing NGC with Goby H+T+D was submitted in response to Popitsch et al (submitted for publication).

While the advantage of our approach may seem modest when compared to CRAM (11% smaller files on average over ten datasets), our approach combines several advantages over CRAM: it provides seamless support for data schema evolution and supports writing alignments before they are sorted (i.e., as required to implement alignment programs). Furthermore, the compression advantage appears strongly dependent on the type of dataset compressed. We see a strong compression advantage for our approaches on RNA-Seq and RRBS datasets (average 21% advantage over 5 datasets). Similarly to CRAM, our approach requires the compression of blocks of HTS alignment data. CRAM uses blocks of one million entries. In our benchmark, we used blocks of 100,000 entries. We found that increasing block size improves the compression efficiency of our approach, however, large blocks also significantly slow down the performance of interactive visualization for the compressed alignments (e.g., visualization in IGV). To our knowledge, CRAM compressed alignments do not currently support interactive visualization, nor are any tools developed to directly analyze data represented in CRAM format. This is in contrast with the H+T+D approach that

Table 3. Benchmark against a BAM baseline.

		Tier 2 Only	Tiers 1+2	
Sample ID	**Kind**	**H+T+D/BAM**	**(CR-BZip2 +H+T+D)/ (FASTQ-Bzip2 + BAM)**	**Multi-tier storage with three alignments: (CR-BZip2 + 3 x H+T+D)/(FASTQ-BZip2 + 3x BAM)**
HZFWPTI	Exome	6.06%	43.35%	23.06%
UANMNXR	Exome	5.85%	43.22%	22.89%
MYHZZJH	RNA-Seq	2.64%	29.00%	13.14%
ZHUUJKS	RNA-Seq	3.14%	39.92%	18.98%
EJOYQAZ	RNA-Seq	12.32%	54.17%	32.45%
JRODTYG	RRBS	22.45%	68.89%	44.92%
ZVLRRJH	Methyl-Seq	24.94%	60.57%	41.47%
XAAOBVT	WGS	7.48%	49.30%	27.34%
UCCWRUX	WGS	6.42%	49.70%	27.26%
HENGLIT	WGS	9.17%	53.78%	31.67%
Average		**10.05%**	**49.19%**	**28.32%**

This table provides compression size ratios calculated for Tier 2 HTS alignments stored with approach H+T+D, with reads (Tier 1+2), or without (Tier 2 only). Reads are stored as PB data compressed with the BZip2 codec (R-BZIP2) or in FASTQ format compressed with Bzip2. The method H+T+D is configured to preserve soft-clips and their quality scores.

doi:10.1371/journal.pone.0079871.t003

supports direct visualization and analysis with a number of HTS software tools [12]. Extending tools to support reading and writing the new file formats can be done by an experienced programmer in one or two days.

Participants to the Sequence Squeeze Competition have presented new techniques to improve the compression of HTS reads data [16]. Because the Goby software can be extended with codecs, it can provide a test bed for developing and testing new read compression techniques, such as those developed for the challenge. We note that improved compression of reads data poses significant practical problems. For instance, codecs must be implemented in a variety of programming languages to make it possible to link with aligner programs written in these languages, because these tools must be able to decompress data on the fly to calculate alignments. For these reasons, our study has initially focused on improving compression of alignment data, while using general compression codecs with robust optimized implementation for read data.

A key feature of the data management approach that we presented here is it ability to support iterative schema extensions. With this approach, file formats do not need to be defined once and for all (such as in an exhaustive format specification), but can be designed iteratively over time. For instance, the Goby read format was designed initially only to support single end reads. When paired end experimental designs became more popular, the read format was extended to store pair sequences and pair quality scores. Another iteration of the design added meta-data to the read file to store information about the sequencing platform and date of sequencing. Importantly, only the features of the file format that are needed at a given time need to be designed and implemented. This makes it easier to design systems that meet evolving requirements, or distributed systems where different version of file formats and program need to interoperate reliably.

The multi-tier data organization that we introduce here makes it possible to reduce some of the computational burden of working with HTS data. Because Tier II alignment file sizes are greatly reduced (by a factor about 10), alignments can be transmitted through the network much faster than would be possible with the BAM format. This is a consequence of both better compression and multi-tier data organization. Multi-tier data organization facilitates visualization of the data as well as collaborative projects that require only access to aligned data (Tier II). For instance, alignments can be loaded through HTTP quickly with IGV. Multi-tier organization also improves the performance of cluster/cloud computing analysis pipelines that require staging of data on compute nodes before computations can take place.

Conclusions

Since Goby formats offer state of the art compression of HTS data and are seamlessly extensible for new applications, we propose that they are strong candidates to replace the FASTQ and BAM formats. Importantly, Goby file formats and software have been leveraged to implement the GobyWeb scalable analysis and data management system [17]. We expect that the structured data compression approaches described here can be applied to a variety of scientific and engineering fields that need to store structured data.

Materials and Methods

The HTS data management approach that we have developed combines novel methods (described in Figure 1 and main text of the manuscript) with a number of standard engineering techniques applied to HTS data:

- Schemas to organize HTS reads and alignments as structured data (see Figure S1 in File S1)
- The Protocol Buffers (PB) middleware to automate reading and writing structured data and to provide flexibility with respect to changes in the schemas [11].
- A storage protocol to store collections of billions of structured messages and support semi-random access to the messages in a collection. The protocol makes it possible to implement Codecs that compress/decompress the PB encoded collections. (Figure S1B in File S1)
- GZip and BZip2 codecs
- A Hybrid codec that provides state of the art compression of alignment data, while retaining the flexibility of PB schema evolution.
- A multi-tier data organization that groups data in tiers according to the most likely use of each kind of data (Figure 2).
- A framework (see http://goby.campagnelab.org/) to support efficient computation with data expressed in these formats.
- A set of tools to work with reads and alignment data in these formats, including

• Tools to import/export alignments from/to the BAM format.

• Tools to import alignments written in the MAF format (produced by the Last aligner [18]).

• Tools to import/export reads from/to the FASTA/FASTQ/CSFASTA formats for single-end or paired-end data [12].

• Integrations into the BWA [9] and the GSNAP [10] aligners to natively load short reads and produce alignment results in these formats.

• Various tools to help process alignments, including sorting, indexing, concatenating several alignments, or merging alignments performed against different reference sequences [12].

• Extensions to the Integrative Genome Viewer (IGV [8]) to load and display the alignment format and display these data along side other data sources in a genomic context.

Protocol Buffers

Protocol Buffers (PB), developed by others, is a software middleware designed "to encode structured data in an efficient yet extensible format" (see http://code.google.com/p/protobuf/). PB offers data representation capabilities similar to the Extensible Markup Language (XML), but that are simpler and significantly more computationally efficient. PB schemas provide a formal data representation language that can express primitive language types as well as complex types of data and their relationship to form messages (equivalent to objects in an OO language or structures in languages such as C). PB provides compilers for a variety of languages that transform schemas into program components suitable to represent data in memory, as well as serialize these data (i.e., write messages to a buffer of bytes) or de-serialize messages (i.e., read a buffer of bytes to reconstitute well-formed messages). Figure S1A in File S1 presents the PB schemas we devised for storing reads and alignments, respectively. The latest schemas can be obtained at http://github.com/CampagneLaboratory/goby/blob/master/protobuf/Alignments.proto and http://github.com/CampagneLaboratory/goby/blob/master/protobuf/Reads.proto. We use Unified Modeling Language (UML) conventions similar to those described in [19] to document the relationships between messages used to store reads or alignments. Rather than encoding complex information in strings (e.g., sequence variations stored as

"CIGAR" strings in BAM), we decompose the information into different PB messages that are simple to process computationally.

Protocol Buffers were initially developed to transmit small messages in client-server environments where software needs to be upgraded asynchronously. As such they provide strong capabilities for schema evolution. For instance, it is possible to add a new message field to a copy of the AlignmentEntry schema, write software that populates the new field, and send data files with this new schema to third parties. Such third parties will be able to use older versions of the software to extract all but the new data structure in the data files. Reading data from the new field will require new software, but third parties can decide if and when they upgrade.

Because PB was initially developed for small messages it would be natively unsuitable for serializing or deserializing the very large collections of messages needed to store billions of reads or alignments. We introduce a storage protocol that addresses this deficiency while retaining the schema evolution capabilities of PB.

Large Collection Storage Protocol

To work around the message size limitation of PB, we introduce the Goby Large Collection Storage Protocol (GLCSP), depicted in Figure S1B in File S1. Briefly, this protocol represents collections of PB messages with N elements as K collections of N/K messages. In the benchmark, we used K = 100,000 so large collections are represented with chunks that contain at most 100,000 messages. Large collections are represented as sequences of chunks. GLCSP supports semi-random access since it is possible to start reading into a GLCSP formatted file at any position and scan until the start of another chunk is encountered. The next chunk found can be decompressed with the codec associated with the registration code found at the start of the chunk (an error to decompress the chunk indicates a false positive delimiter detection, which will be statistically quite infrequent, but needs to be handled appropriately to resume scanning for the next chunk). A termination chunk with 8 successive 0xFF bytes followed by four 0x00 bytes is written immediately before the end of file.

GZip codec

The GZip codec, used since Goby 1.0, simply encodes PB serialized data with GZip compression. The implementation of the GZip codec in Goby uses the standard Java classes: java.util.zip.GZIPInputStream and java.util.zip.GZIPOutputStream.

BZip2 codec

The BZip2 codec, introduced in Goby 2, encodes PB serialized data with BZip2 compression. The implementation of the BZip2 codec in Goby uses the Apache ant implementation: classes named org.apache.tools.BZip2.CBZip2InputStream and org.apache.tools.BZip2.CBZip2OutputStream.

Hybrid Codec

The Goby 2 hybrid codec encodes PB serialized data in two serialized streams: the ACT stream and the Left-Over PB stream. ACT stands for Arithmetic Coding and Template. This new compression approach, described below, consists of compressing collections of K structured messages by serializing fields of the structure messages with arithmetic coding compression. An ACT codec must be implemented for each different kind of schema. At the time of writing this report, we have implemented an ACT codec for the alignment schema described in Figure S1A in File S1. Since PB supports seamless evolution of PB schemas, the hybrid codec must be able to store data that could not be handled by a given ACT implementation (for instance data from fields that have been added to the schema after the ACT implementation was compiled, possibly by a third-party). The hybrid codec stores such data in the Left-Over PB stream. To this end, PB data is serialized and compressed with GZip (Figure 1A).

Arithmetic Coding and Template (ACT) Compression

This approach takes as input a collection of structured messages and produces a stream of bytes with compressed data. This is achieved by considering each field of the messages independently and collecting the successive values of the field when traversing the collection from the first PB message to the last. We reduce each field type to a list of integers. Such lists are compressed as described in section Integer List encoding. Field types are handled as follows. Fields that are recognized by an implementation of ACT (produced against a specific version of the data schema) are removed form the input PB message. Input PB messages that remain non-empty after processing all fields that the codec is aware are written to the Left-Over PB collection output. This simple mechanism suffices to guarantee that older versions of the software do not erase new data fields needed by more recent versions of the software. Fields that are recognized by an ACT implementation are processed as follows, according to their type:

Integer fields. Fields that have a small number of distinct values across all elements of the input collection are written with arithmetic coding list compression (see below). We first introduce the coding techniques used by our integer list compression approach. Fields that follow a uniform distribution (i.e., queryIndex) are written with minimal binary coding.

String fields. String values are converted to list of integers by successively encoding the first character of each string field, then the second, and so on until the length of each string is reached. The length of each string is recorded in a separate integer list.

Floating number fields. Floating numbers (32bits) are stored as their integer representation in an integer list.

Cost of Model Inference

It is important to note that ACT does not eliminate the cost of model inference. Where other approaches incur this cost when presented with a new data file, ACT incurs most the cost once for every data schema, and a much smaller cost for each dataset (for instance when deciding to use run-length encoding for a field of a given dataset). The cost of model inference incurred for the schema is thus amortized over many data files represented with the schema.

Run Length Encoding

Integer lists are scanned to determine if run-length encoding would be beneficial. To this end, a 'lengths' and 'values' list is created from each integer list to code. The 'length' list stores the number of times a given value repeats. The 'values' list simply contains the values of the input list. When the sum of the 'lengths' and 'values' list is smaller than the input list, run-length encoding is used (i.e., we separately write 'lengths' and 'values' lists as described in the section Integer List Compression). Otherwise, the input list is written directly with as a list of integers (see Integer List Compression).

Nibble Coding

Nibble coding is a variable length encoding technique that represents small integers with a small number of bits. We use the Nibble coding implementation provided in the DSI utilities (http://dsiutils.dsi.unimi.it/, Sebastiano Vigna and Paolo Boldi).

The following description is copied from the documentation of the DSI package

"Nibble coding records a natural number by padding its binary representation to the left using zeroes, until its length is a multiple of three. Then, the resulting string is broken in blocks of 3 bits, and each block is prefixed with a bit, which is zero for all blocks except for the last one."

Minimal Binary Coding

A minimal binary code is an optimal code for the uniform distribution and is used to encode query indices (used to link alignment data to read data, see multi-tier organization sections). Briefly, knowing the range of values to be encoded, one can write a natural number in binary code using m bits, where m suffices to encode the maximum value. The value of m is determined by calculating the most significant bit of the maximum value of the list. Minimal binary coding is performed with the DSI utilities. Query indices are written as q-min_q, where q is a query index in a PB collection, and min_q is the minimum value observed in the same collection. The parameter m is detemined as max_q – min_q +1, where max_q is the maximum query index observed in the collection.

Arithmetic Coding

An arithmetic coder is a compression method that yields a code of near optimal length given a specific symbol probability distribution. Arithmetic coders can estimate symbol probabilities adaptively. We use an arithmetic coder implementation derived from that offered by MG4J [20]. However, the Goby implementations of the arithmetic decoder have been optimized for large symbol alphabets (the MG4J decoder has complexity of decoding a symbol O(n), where n is the number of symbol, while the Goby implementation has complexity O(log(n)).

Integer List Compression with Arithmetic Coding

Lists of integers are first inspected to determine if run-length encoding is beneficial. If this is the case, the list is processed as two lists as previously described. Each integer list is then encoded as follows. We write the number of elements of the list with nibble coding, followed by the sign bit (one zero bit if all symbol values are positive, or a one bit if they contain negative values), followed by the number of symbols (nibble coding), the value of each symbol (nibble coding after applying a bijection to map negative integers to natural numbers, when the sign bit was 1). The index of the symbol for each value of the list is then written in sequence using arithmetic coding.

Boolean List Compression

Booleans are converted to the integer value zero or one to produce a list of integers, and further processed as described in the previous section.

Lists of Structured Messages

PB supports messages that contain other structured messages. Goby schemas use this capability to encode sequence variations and links (see Figure S1A in File S1). We compress lists that refer to other messages with as many lists of integers as required to compress each field of the linked message type. Additionally, for each source message, we store the number of elements of the destination message that belongs to the source. For instance, when an AlignmentEntry message includes three SequenceVariation messages, we add the number three to the list that stores the number of sequence variations per entry. We then inspect each SequenceVariation message in the order they appear in the sequenceVariations field and append to the list associated with each field (readIndex, position). Since the traversal order is fixed, this approach can reconstruct both links and linked objects by decoding three sequence variations from the field elements, consuming three readIndex or position values for the entry message.

Template Compression

Template compression is a generalization of the run-length encoding technique for input structured messages. Briefly, for each type of message, we choose the set of fields that will not be included in the template (*non-template fields*). These fields should be chosen as those fields that change the most from one entry to the next. The value of each non-template field is recorded to its respective integer field list and the field is removed from the PB message. In the current implementation, we remove the fields queryIndex, position and toQuality to yield the template. After removing all non-template fields, we are left with a template message. We check if the previously encoded message has the same value as the current template message and if yes increment the number of times the template is to be emitted (we do not emit individual fields for the template in this case). If not, we emit individual fields. A more formal description is given under section 'Algorithm template compression'.

Benchmark datasets

Benchmark datasets were obtained from public databases whenever possible. Accession codes are provided in Table S1 in File S1. The larger files were trimmed to keep only about twenty million reads. Reads that did not map where filtered out. The exact reduced datasets used for the benchmarks can be obtained from http://data.campagnelab.org/

Benchmarks datasets were constructed from public datasets (accession codes: NA12340, NA20766, NA18853, NA19172, GSM675439, GSM721194, ERP000765, ERP000765, SRR065390).

Alignments

Alignments were obtained in BAM format. For samples MYHZZJH and ZHUUJKS, we obtained reads from [21] and realigned against the 1000 genome reference sequence (corresponding to hg19) with GSNAP (version 2012.01.11), allowing for spliced alignments (options for de-novo and cDNA splice detection were enabled). The resulting BAM output is available from http://data.campagnelab.org/.

Benchmark methodology

We developed a set of Bash and Groovy scripts to automate the benchmarks. These scripts are distributed with the benchmark datasets to make it possible to reproduce our results and to assist with the testing and development of new codecs. Scripts copy all data files to local disk before timing execution, and write results to local disks as well, to remove possible variability induced by network traffic. Compression ratio A/B for methods A and B are calculated as the file size obtained when compressing a benchmark file with method A divided by the file size obtained when compressing the same file with method B. Compression ratio are shown as percentage, where 50% indicates that method A compresses the data to 50% the size achieved by the baseline method B. Compression ratios are deterministic, so we do not repeat these measures. Compression speed is measured with the Linux *time* command (using 'real time'), subtracted with the time

taken to read the same input file on the same machine (this time is measured as the time taken to compress the input with a 'null' codec, a codec that writes no output). Decompression speed is measured as the execution time of the Goby compact-file-stats mode. This mode decompresses the successive chunks of an alignment file and estimates and reports statistics about the entries in the alignment. Compression and decompression speeds are largely deterministic, varying only by a few percentage from run to run. We omitted standard errors for clarity because repetitive runs showed virtually no variation in our test environment.

CRAM parameter settings

We used three parameter settings in our evaluation of CRAM 2.0. The settings are called S1, S2 and S3, where S1 is the most lossy format. These settings are as follows: S1: the most lossy compression, does not keep soft clips nor unmapped placed reads (default command line arguments with logging at the INFO level: -l INFO). Setting S2: intermediate lossy compression, keeps soft clips and unmapped placed reads, keeps quality scores for mutations and insertion deletions (command line arguments: -l INFO --lossy-quality-score-spec N40). Setting S3: lossless compression, like S2 but also keeps quality scores for the complete reads, the BAM attribute tags and preserve unmapped reads (command line arguments: -l INFO --capture-all-tags --lossy-quality-score-spec N40-R40-U40).

Goby parameter settings

BAM files were converted to Goby file format with the sam-to-compact tool. Files were initially written with the GZIP codec, and re-compressed with each codec using the concatenate-alignment tool (see benchmark scripts). We used the following parameters to measure compression with the ACT approach:

To create files comparable with CRAM2.0 S2, we preserved soft-clips and quality scores over variations (option --preserve-soft-clips of the Goby sam-to-compact tool). To create files comparable with CRAM2.0 S3, we preserved soft-clips and quality scores over the entire read (options --preserve-soft-clips --preserve-all-mapped-qualities of the Goby sam-to-compact tool).

The following settings were used with the concatenate-alignment tool:

- Compression with gzip codec: -x MessageChunksWriter:codec = gzip -x AlignmentWriterImpl:permutate-query-indices = -false -x AlignmentCollectionHandler:ignore-read-origin = true --preserve-soft-clips.

- Compression with bzip2 codec: -x MessageChunksWriter:codec = bzip2 -x AlignmentWriterImpl:permutate-query-indices = -false -x AlignmentCollectionHandler:ignore-read-origin = true

- Compression with ACT H approach: -x MessageChunksWriter:codec = hybrid-1 –x MessageChunksWriter:template-compression = false –x AlignmentCollectionHandler:enable-domain-optimizations = true –x AlignmentWriterImpl:permutate-query-indices = false -x AlignmentCollectionHandler:ignore-read-origin = true

- Compression with ACT H+T+D approach: -x MessageChunksWriter:codec = hybrid-1 –x MessageChunksWriter:template-compression = true –x AlignmentCollectionHandler:enable-domain-optimizations = true -x AlignmentWriterImpl:permutate-query-indices = false -x AlignmentCollectionHandler:ignore-read-origin = true

Multi-tier data organization

A critical advantage of a multi-tier file organization is the ability to study a dataset with multiple alignment methods. With single file organization (e.g., BAM or lossless CRAM), multiple analyses result in duplicating data. Indeed, projects that produce BAM alignments typically already have read data in FASTQ format. The read data are duplicated in each new BAM file that a project produces with a different alignment approach. With multi-tier organization, only alignments are stored for each analysis, further increasing storage efficiency for projects that align reads with multiple methods. Table 2 indicates that multi-tier organization can yield substantial storage savings when compared to a FASTQ/BAM storage scheme. It is worth noting that each schema includes fields that provide meta-data to assist tracking relations between data files (See Figure S1A in File S1, MetaData message in the read schema, and ReadOriginInfo in the alignment schema).

Preserving read trackability

Multi-tier organization must preserve the identity of a read across tiers. This is necessary to link back specific analysis results to raw data. In the BAM format, read-names link alignment results to the primary read data. This solution is effective, but wasteful: it requires storing long strings of characters ($\sim$15 characters in most current datasets, or 6+ bytes at least if unique integers are written as strings) whose only function is to maintain read identity. In the Goby multi-tier organization, read identity is maintained with an integer index. This index tracks read identity during the entire life cycle of HTS data. Special considerations must be taken to guarantee that preserving this index during the data life-cycle does not degrade compression performance. We discuss these methods in the next section.

Query Index and Permutations

Goby maintains the link between alignments and primary read data with an integer, called a query index (See Figure S1A in File S1, ReadEntry message type). When an alignment program processes a Goby reads file, the query index field of the read entry is written to the alignment entry to preserve the link to the raw data (see Figure 1B, AlignmentEntry). Sorting the alignment will result in shuffling query indices, and can seriously degrade compression performance of a sorted alignment (because compression of a sequence of uniformly distributed 32 bit integers requires 32 bits per integer). We avoid this problem by permuting the original query indices to small indices that monotonically increase in genomic order. The small indices are still uniformly distributed, but in a much reduced range, and therefore can be compressed more effectively (with the minimal binary coding method). Permutations are written to disk in a specialized data structure that makes it possible to retrieve the original query index corresponding to any small index (stored in a.perm file in Tier III). Permutation files are only necessary for those applications that need to track read indices back to primary read data. Tier II alignments can be used in isolation or together with data from Tier III, depending on the needs of the application. We note that CRAM did not address the issue of storing read identity because (1) CRAM can only compress sorted alignments, (2) Converting a BAM file to CRAM does not maintain a mapping between read index and read name.

Compression/decompression fidelity

We routinely test whether the H+T and H+T+D compression approaches implemented in Goby are able to recover the input dataset after decompression. To this end, the Goby source code includes a regression test suite that compares alignment datasets after round-trip compression/decompression. The regression suite is run after each modification of the project source code and helps

control that current and future versions of the software have perfect data fidelity over representative test datasets.

Algorithm Coding and Template compression

Inputs:
A collection of PB messages in a GLCSP chunk
Outputs:
A number of integer lists, one for each field of the PB messages
Init:
T = nil
C = 0
for each message m in PB chunk:
for each non template field g of m:
emit g to integer list corresponding to g field
remove g from m
end
if g equals T then
C+ = 1
continue with next message
else
emit C to integer list corresponding to message count field
for each field e of m:
emit e to integer list corresponding to e field
end
T = m
C = 1
end
end

Supporting Information

File S1 Supporting Tables and Figures. This file contains Table S1, Table S2 and Figure S1. Table S1: Details of the Benchmark Datasets; Table S2: General Compression Benchmark relative to GZip; Figure S1: Structured data schemas and Large Collection Storage Protocol.
(PDF)

Acknowledgments

We thank Dr. Thomas Wu (Genentech Inc.) for help with the GSNAP implementation, and Dr. Olivier Elemento for critical feedback on earlier revisions of this manuscript.

Author Contributions

Conceived and designed the experiments: FC. Performed the experiments: FC KCD. Analyzed the data: FC KCD NC. Contributed reagents/materials/analysis tools: NC JTR JPM. Wrote the paper: FC. Read and approved the final manuscript: FC KCD NC JTR JPM.

References

1. Mangone M, Manoharan AP, Thierry-Mieg D, Thierry-Mieg J, Han T, et al. (2010) The landscape of C. elegans 3'UTRs. Science 329: 432–435.
2. Agrawal N, Frederick MJ, Pickering CR, Bettegowda C, Chang K, et al. (2011) Exome sequencing of head and neck squamous cell carcinoma reveals inactivating mutations in NOTCH1. Science 333: 1154–1157.
3. Shearstone JR, Pop R, Bock C, Boyle P, Meissner A, et al. (2011) Global DNA demethylation during mouse erythropoiesis in vivo. Science 334: 799–802.
4. Mardis ER (2011) A decade's perspective on DNA sequencing technology. Nature 470: 198–203.
5. Cock PJ, Fields CJ, Goto N, Heuer ML, Rice PM The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic Acids Res 38: 1767–1771.
6. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, et al. (2009) The Sequence Alignment/Map format and SAMtools. Bioinformatics 25: 2078–2079.
7. Hsi-Yang Fritz M, Leinonen R, Cochrane G, Birney E (2011) Efficient storage of high throughput DNA sequencing data using reference-based compression. Genome Res 21: 734–740.
8. Robinson JT, Thorvaldsdottir H, Winckler W, Guttman M, Lander ES, et al. (2011) Integrative genomics viewer. Nat Biotechnol 29: 24–26.
9. Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 25: 1754–1760.
10. Wu TD, Nacu S (2010) Fast and SNP-tolerant detection of complex variants and splicing in short reads. Bioinformatics 26: 873–881.
11. Varda K (2008) Protocol Buffers. Available: http://code.google.com/p/protobuf/.
12. Campagne F, Dorff K, Chambwe N, Robinson JT, Mesirov JP, et al. (2012) Compression of structured high-throughput sequencing data. Preprint at arXiv. Available: http://arxiv.org/abs/1211.6664.
13. Burrows M WD (1994) A block-sorting lossless data compression algorithm. Digital Equipment Corporation.
14. Effros M. PPM Performance with BWT Complexity: A Fast and Effective Data Compression Algorithm; 2000; Washington, DC, USA.
15. Popitsch N, von Haeseler A (2012) NGC: lossless and lossy compression of aligned high-throughput sequencing data. Nucleic Acids Res 41: e27.
16. (2012) Sequence Squeeze Competition. Available: http://www.sequencesqueeze.org/.
17. Dorff KC, Chambwe N, Zeno Z, Shaknovich R, Campagne F (2012) GobyWeb: simplified management and analysis of gene expression and DNA methylation sequencing data. arXiv http://arxivorg/abs/12116666.
18. Kielbasa SM, Wan R, Sato K, Horton P, Frith MC (2011) Adaptive seeds tame genomic sequence comparison. Genome Res 21: 487–493.
19. Skrabanek L, Murcia M, Bouvier M, Devi L, George SR, et al. (2007) Requirements and ontology for a G protein-coupled receptor oligomerization knowledge base. BMC bioinformatics 8: 177.
20. Boldi P, Vigna S. MG4J at TREC 2005. In: Buckland EMVaLP, editor. Special Publications; 2005. NIST.
21. Pickrell JK, Marioni JC, Pai AA, Degner JF, Engelhardt BE, et al. (2010) Understanding mechanisms underlying human gene expression variation with RNA sequencing. Nature 464: 768–772.